Aardvark: A Concurrent Authenticated Dictionary with Short Proofs

Derek Leung, Yossi Gilad, Sergey Gorbunov, Leonid Reyzin, Nickolai Zeldovich
Algorand, Inc.
{derek,yossi,sergey,leo,nickolai}@algorand.com

***Abstract*—We design Aardvark, a novel authenticated dictionary backed by vector commitments with short proofs. Aardvark guarantees the integrity of outsourced data by providing proofs for lookups and modifications, even when the servers storing the data are untrusted. To support high-throughput, highly-parallel applications, Aardvark includes a versioning mechanism that allows the dictionary to accept stale proofs for a limited time.**

We apply Aardvark to the problem of decoupling storage from transaction verification in cryptocurrencies. Here networking resources are at a premium and transmission of long proofs can easily become the dominant cost, with multiple users reading and writing concurrently.

We implement Aardvark and evaluate it as a standalone authenticated dictionary. We show that Aardvark saves substantial storage resources while incurring limited extra bandwidth and processing costs.

I. Introduction

Ensuring the integrity of vast record collections constitutes a key problem in computer security. In 1991, Blum et al. [3] described a scenario where a trustworthy computer executes programs on data that resides in untrusted storage. To prevent the untrusted storage from tampering with the data, they introduced a data structure, now known as a two-party *authenticated dictionary* [12], [18], which allows the trusted computer to ensure the correctness of accesses and changes to data. The computer stores only a small *commitment* to the data. Using *proofs* provided by the untrusted storage for each read or write operation, the computer can verify correctness of every read and correctly update its commitment for every write.

Today, with the ubiquity of distributed systems that process large amounts of data, it has become attractive to decouple trust in the correctness of updates from the cost of backing storage itself. *Cryptocurrencies* are one pertinent system. Here, any user can send money to another user by submitting a transaction over the records kept by the system. Validators authenticate these transactions, check their validity against the system records, serialize them using a fault-tolerant protocol, and update the system records to reflect the transactions. There have been several proposals to make validators *stateless* — i.e., enable validators to verify transactions and update records without storing the records themselves. In these proposals, the validators store and update only a small commitment to the records, while untrusted archives back their physical storage. Each transaction comes with a proof of validity of the necessary read and write operations.

This paper presents Aardvark, an authenticated dictionary which enables such a division of labor and, in particular, scales up to many users and many transactions per second. Aardvark supports highly-distributed applications where many untrusted clients concurrently execute transactional read and write operations (such as in a cryptocurrency). This deployment environment raises several key problems.

First, network bandwidth is at a premium because transactions must be observed by many machines, and transmitting proofs required by each transaction over the network becomes a bottleneck to high throughput. Aardvark introduces a novel data structure backed by a *pairing-based vector commitment scheme* where all proofs are roughly $10\times$ shorter than standard Merkle-tree commitment schemes. This data structure is the first of its kind that supports a true authenticated dictionary interface; specifically, by explicitly tracking gaps in the dictionary's key space, Aardvark enables short proofs both of key membership and of key nonmembership.

Second, the ability for untrusted clients to allocate and deallocate storage raises the need for efficient garbage collection. Aardvark's enforces tight upper bounds on resource use even under adversarial access patterns. Aardvark supports quick reclamation of deallocated space by decoupling the logical ordering of keys in the dictionary, which expedites proofs of nonmembership, from their ordering in commitments, which eagerly compacts data into contiguous vectors.

Third, scaling to many users requires the system to behave well under a substantial amount of concurrency. When the dictionary's state changes, the commitments also change, invalidating proofs for transactions. Executing transactions with old proofs requires updating the proofs against the new commitment, which involves a non-trivial computational cost in pairing-based vector commitment schemes. This cost must be paid for every pending transaction and is highest when the system is under a maximal load, exacerbating the problem. Instead of attempting to reduce these costs, Aardvark avoids them entirely through a new versioning mechanism for the dictionary. This mechanism enables the system to efficiently check stale proofs against an old dictionary state by supporting copy-on-write semantics, implemented through small deltas on the dictionary keys.

Contributions To summarize, this document establishes the following contributions.

1) An authenticated dictionary, backed by vector commitments, which may be efficiently used to prove key nonmembership and possesses tight upper-bounds on excess resource use even under adversarial access patterns (§V).

2) A versioning mechanism for this dictionary, so that it supports concurrent transaction execution (§VI).
3) An implementation of our design and an evaluation of its performance as a standalone authenticated dictionary showing that it is practical (§VII).

II. Related Work

We provide context for the design of Aardvark by comparing it to (1) other authenticated dictionaries and (2) to other storage systems which support stateless cryptocurrencies.

Authenticated Dictionaries. Most constructions of authenticated dictionaries [12], [18], [20] are based on some variant of the Merkle hash tree [15]. (There are exceptions that require verifiers to keep secrets from the untrusted storage—e.g., [19]; we do not consider them here, as cryptocurrency validators are public and thus cannot be trusted with secrets.) Such constructions add substantial network bandwidth costs to the cryptocurrency, which may become a bottleneck in transaction throughput [9]. Merkle proofs at 128-bit security for an authenticated dictionary of just 100 000 entries are over a half a kilobyte in size. In contrast, a small transaction (e.g., transferring money in a cryptocurrency) may be as short as 100B , so proofs using Merkle trees impose a bandwidth overhead of over 10x if the transaction involves accessing two keys in the dictionary (e.g., source and destination accounts in the cryptocurrency context). As the number of keys in the dictionary grows, this overhead increases; for a billion accounts, the overhead becomes about 20x.

Boneh, Bünz, and Fisch [5, Sections 5.4, 6.1] propose the first authenticated dictionaries that require no verifier secrets and do not use Merkle trees; instead, they rely on factoring-based (or class-group-based) accumulators and vector commitments. They also suggest (but do not investigate the costs of) using such dictionaries to implement stateless clients. The proof length in their construction is over 1KB and thus comparable to Merkle trees (at 128-bit security). Moreover, to avoid computationally expensive modular exponentiations (with exponents of length at least 16,384), it is necessary to store and update two accumulators, thus doubling the proof length. However, it is possible to batch proofs for multiple operations into a single constant-size proof.

Aardvark is the first system that uses pairing-based vector commitment schemes to build authenticated dictionaries. A vector commitment (introduced in [7], [14]) implements an authenticated array. Aardvark uses pairing-based vector commitments of Libert and Yung [14]. At 128-bit security, proofs in this scheme take up only 48B *regardless of the number of keys*, or $10\times$–$20\times$ less than Merkle proofs, and comparable to the size of a transaction. We are able to retain these short proofs in Aardvark. We should note that all pairing-based vector commitment schemes have public parameters (needed by provers and verifiers) that require trusted generation and are at least linear in the size of the vector. Thus, Aardvark also requires public parameters; however, we avoid linear-size public parameters in Aardvark by storing data as multiple shorter vectors (as a result, our commitment is not constant-size—instead, there is a tradeoff between commitment size and public-parameter size). We note that multiple proofs in Aardvark can be aggregated into a single constant-size proof, as shown in [13].

Stateless cryptocurrency validation. Cryptocurrency designs can be broadly grouped into two models: unspent transaction outputs (UTXO) model and account-based model.

In the UTXO model, the state of the system at a given time is characterized by a set (namely, the set of transaction outputs that have not yet been used as inputs to new transactions). To allow for stateless validation in such a design, one needs a cryptographic accumulator (which is an authenticated version of the dynamic set data structure). Designs based on Merkle trees were proposed for this purpose in a number of works [6], [8], [11], [17], [16], [23], [24], [26]. Like Merkle-based authenticated dictionaries, Merkle-based accumulators suffer from the proof length problem, although [11] proposes some optimizations based on proof aggregation and local caching of the tree nodes (thus adding some state back to the clients). Some other cryptographic accumulators are suitable as well; a discussion of options is available in [10]. In particular, a design based on groups of unknown order (such RSA and class groups) is proposed in [5, Section 6.1] (each proof in this design is over 1KB, but it allows for aggregating proofs for multiple transactions).

In contrast to the UTXO model, in the account-based model, the state of the system at a given time is characterized by a map (dictionary) from account identifiers to account states. As already mentioned, such authenticated dictionaries are designed and proposed for cryptocurrency uses in [20] and [5], but required relatively long proofs and were not evaluated in detail in the context of a cryptocurrency.

The most detailed and elegant design of an account-based cryptocurrency with stateless validation called Edrax is provided by Chepurnoy et al. [8]. Edrax assigns sequential integers as account identifiers, thus simplifying the necessary data structure from an authenticated dictionary to an authenticated array. This simplification enables Edrax to use vector commitments directly. Because Edrax represents the entire state of the system as a single vector, it has to handle the problem of linear-sized public parameters for pairing-based vector commitments. It does so by designing a vector commitment scheme in which only logarithmic-sized pieces of public parameters are needed for various operations; these are provided at the right time by the right parties. Tomescu et al. [25] improve this vector commitment, achieving constant-size proofs and requiring only constant-size portions of the public parameters for the relevant operations (moreover, these portions are verifiable, thus reducing the need for additional interaction); they also refine the overall system design.

In order to assign sequential identifiers, Edrax requires every new user to register via an initialization transaction. In particular, in order to send money to a user, the sender needs to make sure the user is registered first, in order to learn that user's identifier. The number of sequential identifiers, and thus initialization transactions, is limited; it thus becomes necessary to prevent denial-of-service attacks by multiple initialization transactions (see [25, Section 4.2.4] for possible approaches). By using authenticated dictionaries rather than authenticated arrays, Aardvark able to avoid this initialization transaction.

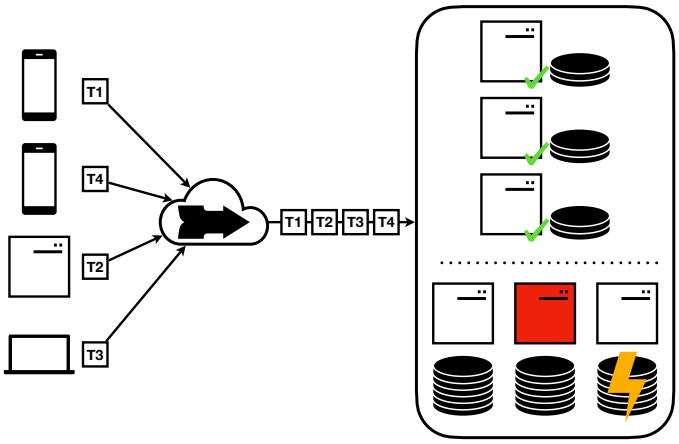

Fig. 1. Aardvark's architecture. On the left, untrusted clients issue transactions in an arbitrary order. These transactions are serialized and given to Aardvark on the right, which is composed of reliable validators and unreliable archives. Note that the serialization order of the transactions is the same for both validators and archives.

In particular, in Aardvark it is possible to transact with keys that have not yet registered with the system.

Unlike Aardvark, EDRAX does not address versioning that is necessary because a transaction's proof is obsoleted by blocks produced between the time the transaction gets issued and the time it is being processed. There are other design differences between Aardvark and EDRAX: unlike Aardvark, EDRAX has every account holder keep track of their own proof; and, relying on the additive homomorphism of vector commitments, EDRAX does not require proofs for the state of a recipient account (this works for simple transactions that add a fixed numerical amount to a balance, but not for more complex transactions).

III. OVERVIEW

Aardvark is a *dictionary*, that is, a data structure that associates *keys* to *values*. Clients interact with Aardvark by issuing operations against the dictionary to read the value associated with a key, associate a new value to a key, or to disassociate a key from any value. All operations are grouped into atomic units called *transactions*, which are defined below.

Aardvark assumes that clients issue these transactions to some *serialization service*, which assigns a unique sequence number to each transaction. One example of such a service is a fault-tolerant protocol executed by validators of a cryptocurrency. Aardvark then executes transactions in order of their sequence number. If a transaction succeeds, it results in an updated dictionary state.

Servers in Aardvark are partitioned into two groups: *validators* and *archives*. Validators have access to a limited amount of durable, transactional storage but are mostly trusted to execute the transaction faithfully.[1] In contrast, archives have access to substantial amounts of storage but may deviate arbitrarily from correct behavior. For instance, an archive machine may lose power and fail to respond to queries, or it might be compromised by an attacker and produce malicious responses. This system architecture is illustrated in Figure 1.

A. Goals

Under these assumptions, Aardvark attempts to achieve the following goals.

- *Correctness*: All successful transactions, when executed, result in an updated dictionary state consistent with the operations' semantics. No operation in any unsuccessful transaction is executed.
- *Transaction Serialization*: When a transaction executes, it observes the effects of all successful transactions preceding it in the serialization order and none of the effects of all transactions following it.
- *Expressiveness*: Transactions are expressive enough to support the requirements of the calling application.
- *Performance*: Many transactions may be executed at high throughput.
- *Efficiency*: Transactions are executed without inordinate resource overheads.
  - *Storage*: Validators benefit from substantial reductions in storage costs. Even under adversarial modification patterns, storage costs for a validator should be upper-bounded by a fixed, small, and constant fraction of the size of all key-value mappings.
  - *Network*: Additions to transaction size are small.
  - *Computational*: Few additional computational costs are required to process transactions. Processing costs may be parallelized.
- *Archive Independence*: Clients can choose to *monitor* keys in lieu of trusting an archive for availability. Clients may issue transactions which exclusively interact with monitored keys even if all archives are faulty.

B. Transactions

Clients interact with the dictionary by submitting *transactions*, which are sequences of *operations* guaranteed to be *atomic*. In particular, Aardvark guarantees that if a transaction is executed, either all operations execute, or none do. All transactions are of the following form:

$$\text{Transaction}(\{k_1, k_2, \ldots, k_m\}, \\ [op_1, op_2, \ldots, op_n])$$

Specifically, a transaction specifies a set of m keys and a list of n operations. A key is a bitstring which identifies a single record. The list of operations represents a deterministic program which may contain the following special operations for interacting with the system state:

$$\text{Get}(k_i) \to v$$
$$\text{Put}(k_i, v)$$
$$\text{Delete}(k_i)$$
$$\text{Assert}(cond)$$

[1] Not all validators need to be trusted for correctness. Applications such as a cryptocurrency may execute a fault-tolerant protocol that resists a fraction of misbehaving validators.

Get returns the value associated with the given key in the key-value store, and Put associates a new value with a key in the key-value store, overwriting any old value. Delete removes a key from the store. Any key passed to Get, Put, or Delete must be in the key set specified by the transaction. The Assert operation, parameterized by a boolean condition, causes the entire transaction to fail if its condition is not met.

On intitialization, users specify a *default value* for the key-value store. If a key is absent from the store, Get returns that default value.

For example, suppose `alice` and `bob` denote keys for two cryptocurrency balances, which have a default value of 0. Then a transfer of p units of money from `alice` to `bob` may be implemented through the following transaction, in pseudocode:

```
with Transaction(alice, bob) as txn:
    a = txn.Get(alice)
    b = txn.Get(bob)
    txn.Assert(a >= p)
    if a-p == 0:
        txn.Delete(a)
    else:
        txn.Put(a, a-p)
    txn.Put(b, b-p)
```

C. System Design

Operations in Aardvark must be carried out by validators, but the actual account data is backed by archives. For validators to correctly execute their operations, they must receive appropriate *transaction contexts* from the archives. Since archives may be compromised, validators must authenticate these contexts. It follows that transaction operations involving Aardvark keys map directly to validator operations with an extra untrusted context argument σ:

$$\begin{aligned}&\text{Get}(k, \sigma_k) \to v\\ &\text{Put}(k, v, \sigma_k)\\ &\text{Delete}(k, v, \sigma_k, \sigma_k^*)\end{aligned}$$

(Delete requires more context than Get or Put, which we explain in §V.) Likewise, archives must generate these contexts, and Aardvark requires them to implement the interface

$$\begin{aligned}&\text{Read}(k) \to \sigma_k\\ &\text{ReadPred}(k) \to \sigma_k^*\end{aligned}$$

Finally, Aardvark grants clients the ability to *monitor* keys so that they can continue performing transactions on them even if no archives function correctly. This operation consists of a single functionality, SyncContext, which updates a piece of context given an update to the dictionary state.

To explain the design of Aardvark, this paper proceeds in two phases. First, §V explains how Get, Put, and Delete are implemented on *validators* given a particular snapshot of the dictionary state, in order to illustrate how the dictionary maintains its invariants given a single transaction. Second, §VI explains how the system executes multiple transactions concurrently; in particular, it develops a versioning scheme for the dictionary and describes how archives and clients produce proofs against different snapshots of the dictionary state.

Before we describe the details of our implementation, we first review the cryptographic primitives available to us in vector commitments.

IV. Background: Vector Commitments

A vector commitment (introduced in [14] and formalized in [7]) cryptographically commits to an array V of data, of size fixed at initialization. The Commit procedure produces a short value c out of V. Given V and an array index i, it is possible to produce a proof ρ that the ith entry in the array is v_i. This proof can be verified against c. The *binding* property of the commitment ensures that it is computationally infeasible prove any other value than v_i. When a value in V change, both the commitment and the proofs for all values can be efficiently updated. We now provide a more formal description.

A. Interface

The following summarizes the black-box interface we expect of a vector commitment scheme:

$$\begin{aligned}&\text{ParamGen}(B) \to pp\\ &\text{Commit}(pp, V) \to c\\ &\text{Open}(pp, V, i) \to v_i, \rho\\ &\text{Verify}(pp, c, i, \bar{v_i}, \rho) \to \text{ok}\\ &\text{CommitUpdate}(pp, c, (i, v_i, v_i')) \to c'\\ &\text{ProofUpdate}(pp, \rho, j, (i, v_i, v_i')) \to \rho'\end{aligned}$$

ParamGen generates parameters that can be used to commit an array of size B. These parameters must be generated by a trusted procedure (for example, through multiparty computation) that does not reveal the secrets used. One set of parameters can be used for multiple arrays of size B each.

Commit creates a cryptographic commitment to an B-sized vector of variable-length data.[2] Commit is deterministic: if v_1 and v_2 are two B-sized vectors and $v_1 = v_2$, then $\text{Commit}(v_1) = \text{Commit}(v_2)$. We rely on the determinism of Commit to optimize proof generation.

Open (like Commit, deterministic) creates a cryptographic proof that a particular vector element v_i (present at index $i < B$ in the vector v) is committed to in $\text{Commit}(V)$. Verify checks that a proof is valid.

CommitUpdate returns $\text{Commit}(V')$, where V' is obtained from V by changing the ith entry to v_i'. Similarly, ProofUpdate returns the cryptographic proof $\text{Open}(pp, V', j)$ for element j in this V'. Note that CommitUpdate and ProofUpdate do not require knowledge of V, unlike Commit and Open.

B. Vector Commitments of Libert and Yung

As stated before, our particular choice of commitment is motivated by our efficiency requirements. In particular, bandwidth costs are one factor that limit system throughput. We use vector commitments from [14]. They have the smallest known proofs: only 48 bytes for 128-bit security, or about an order of magnitude shorter than Merkle proofs for reasonable data sizes

[2] Variable-length data can always be converted to fixed-length by hashing before applying Commit.

TABLE I. LIBERT-YUNG VECTOR COMMITMENT EFFICIENCY

Operation	Latency
Commit	36ms
Open	36ms
Verify	2.8ms
CommitUpdate	200μs
ProofUpdate	200μs

and security parameters. Moreover, as shown by Gorbunov et al. [13], multiple proofs (even for different commitments) can be aggregated into a single 48-byte value and can be verified at once.

These vector comments do, however, require, parameters pp that are linear in B (see [13, Section 4.1] for a discussion of how such parameters can be generated).

Another important cost on the critical path is the computational overhead of interacting with proofs. While the proof generation work is done by untrusted archives and can be divided amongst many untrusted machines (e.g., on some cloud provider with cheap storage), proof verification must be done by all validators, so it lies on the critical path.

To manage computational costs, Aardvark must allow proof verification to execute in parallel on independent threads and minimize the number of required verification operations. We take advantage of lazy execution and batching of operations to further mitigate costs. We refer the reader to §VI for these techniques.

Our scheme may trade off its storage savings against computational overhead by tuning B. We set $B = 1000$. Note that verification time does not vary in B, but all other operations do.

Efficiency. To provide context and motivation for the rest of Aardvark's design, we benchmark key operations in our implementation of these vector commitments on a `c5.metal` Amazon EC2 machine with $B = 1000$. Table I provides an overview of these costs.

V. AUTHENTICATED DICTIONARY DESIGN

Aardvark maintains an authenticated mapping from keys, which are arbitrary bitstrings, to the values associated with those keys. The space of possible bitstrings of any given size is exponentially large in the key size and thus much bigger than the size of the mappings. To leverage the compactness of vector commitments, data must be contiguously packed. As a result, we must find an efficient encoding of the dictionary's mapping in arrays.

One direct approach for maintaining such a mapping is storing all (key, value) entries in an arbitrary order in arrays of size B each (recall that B is the vector size parameter specified in IV-A). To prove that a key is associated with a given value, the prover runs Open on the relevant array and index, proving the (key, value) map entry. Modifying the value associated with a given key involves simply running CommitUpdate. Less obvious is how a prover shows that a key does *not* exist. What if the prover lies about a key's absence in a mapping? If we solve this problem, we can also ensure that every key appears at most once, because as part of the insertion proof, the prover would show that the key is absent from the dictionary before insertion.

At a high level, Aardvark achieves this invariant by utilizing two distinct orderings for the key-value pairs. First, Aardvark commits to its keys in an arbitrary *sequential* ordering of keys in the *slots* of the length-B vectors. Because this ordering is arbitrary, we can always insert to the tail end of the last vector, and easily compactify upon delete, thus minimizing both the number of cryptographic operations and the overhead of empty space. Second, Aardvark commits to an independent *lexicographic* ordering of its keys, allowing a prover to show that a key does not exist in the dictionary. We do so by storing, with each key, its lexicographic successor.

Initialization. Our dictionary is initialized with vector commitment parameters, as specified in the previous section. The dictionary is parameterized with a bucket size B, which is similarly fixed at initialization. In addition, it must be initialized with a single key and value, as it must maintain the internal invariant that at least one key is present at any time.[3]

A. Contiguous Slot Packing

One challenge of implementing our dictionary is that users may wish to insert many arbitrary keys and then later remove these keys. Aardvark must efficiently scale up and down relative to these modifications, which could be adversarial in nature. We propose an allocation scheme for Aardvark which guarantees a limited worst-case overhead regardless of access pattern.

To effectively store an arbitrary distribution of keys in a sparse representation, our dictionary hashes each key-value pair and places these hashes into consecutive *slots*. These slots are partitioned into contiguous *buckets* of size B. Each bucket holds B slots; the lth bucket holds items in the range $[Bl, Bl + B)$. We will denote the data in lth bucket by $D[l]$.

For each l, archives store $D[l]$, while validators store $\text{Commit}(D[l])$. Validators also store the total size of the dictionary, s. Let D denote the sequence of all $D[l]$, and define the *dictionary digest* to be the sequence of all commitments to all buckets. To modify a single slot, an archive computes the absolute index of the slot in the slot ordering, along with the proof of the slot's current value at that index. This data allows validator to modify the slot in place:

```
# db is a handle for validator storage
# db.commits[l] fetches commitment l
# i is the index of the slot
# old is the old slot value
# new is the old slot value
# pf is a vector proof for old at i
def slot_write(db, i, old, new, pf):
    bucket, off = i/B, i%B # floor division
    c0 = db.commits[bucket]
    if not Verify(c0, off, old, pf):
        raise Exception("proof is invalid")
    c1 = CommitUpdate(c0, (off, old, new))
    db.commits[bucket] = c1
```

[3] In a general setting, this invariant may be achieved by adding a sentinel key to the dictionary which is never deleted.

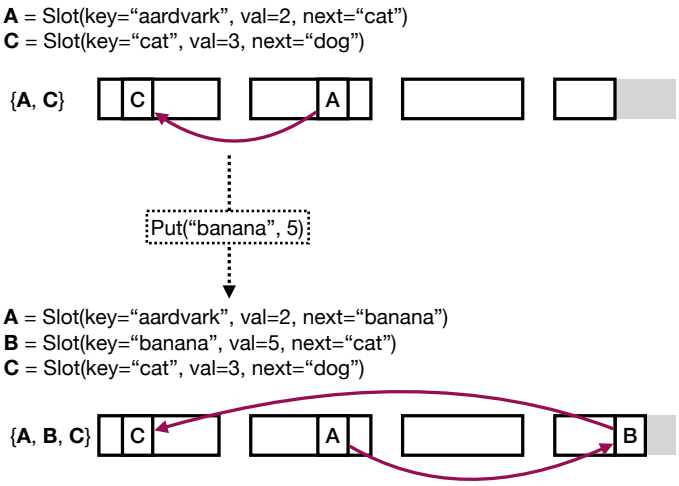

Fig. 2. Key insertion in Aardvark consists of the following steps. (1) The key is inserted into the last slot of the dictionary. (2) Its predecessor is verified, and the new slot's next pointer is attached to the predecessor's next pointer. (3) Its predecessor's next pointer is updated to the inserted key. Observe that the operations shown here require only one update operation on a vector commitment (updating the predecessor's next pointer) since the tail is stored by validators and not committed to.

To enforce tight bounds on the space overhead of the dictionary, Aardvark's packing scheme maintains the following invariant: all validators must store at most $\lceil s/B \rceil$ vector commitments plus a small, constant overhead cost. When a new element is inserted into the dictionary, it is added to the last slot in the dictionary (at s); when an element is deleted from the dictionary, it is overwritten with the element at the last slot (at s), and the last slot is cleared from the dictionary:

```
# db.size is the total number of slots
def slot_append(db, new):
    if db.size%B == 0:
        db.commits.append(Commit([new]))
        return
    bucket, off = db.size/B, db.size%B
    c0 = db.commits[bucket]
    c1 = CommitUpdate(c0, (off, 0, new))
    db.commits[bucket] = c1

# slot_s is the slot at index s=db.size
# pf_s is a vector proof for slot_s at s
def slot_truncate(db, i, old, pf,
                  slot_s, pf_s):
    bucket_l, off_l = db.size/B, db.size%B
    c_l = db.commits[bucket_l]
    if not Verify(c_l, off_l, slot_s, pf_s):
        raise Exception("proof(s) is invalid")
    bucket, off = i/B, i%B
    db.size = db.size - 1
    if db.size%B == 0:
        db.commits.pop(-1) # drop last item
    if not Verify(c0, off, old, pf):
        raise Exception("proof(i) is invalid")
    c0 = db.commits[bucket]
    c1 = CommitUpdate(c0, (off, old, slot_s))
    db.commits[bucket] = c1
```

B. Authenticating Reads and Writes

To prove that a key is associated with a particular value, an archive locates that key in its bucket and computes a proof of that key's existence with Open. For validators, overwriting that key is straightforward: invoke CommitUpdate on the old commitment to that bucket, using the new value along with the key and the old value, which are contained in the proof of membership for that operation:

```
# val is the value to be written
# ctx contains the context for the key
#   ctx.index contains the key's index
#   ctx.slot contains the slot at ctx.index
#   ctx.pf proves that ctx.slot is at ctx.index
def Put_modify(db, val, ctx):
    old = ctx.slot
    new = ctx.slot
    new.val = val
    slot_write(db, ctx.index, old, new,
               ctx.pf)
```

We now come to the lexicographic ordering. Proving that a key does not exist in the dictionary involves adding an extra field *next* to each value in the dictionary. This field is committed in the buckets along with the rest of the value. For any key k, $next_k$ holds the successor of that key, denoted succ, which is defined as follows:

1) If k is the only key in the dictionary, $\text{succ}(k) = k$.
2) If k is the largest key in the dictionary, $\text{succ}(k)$ is the smallest key in the dictionary.
3) Otherwise, $\text{succ}(k)$ is the smallest key in the dictionary larger than k.

To prove that a key does not exist in the dictionary, an archive locates the key's *predecessor*. The predecessor of a key k is defined as the largest key in the dictionary which is smaller than k in lexicographic ordering, with the exception that the largest key in the dictionary is the predecessor of the smallest key in the dictionary and all keys smaller than that key. The archive then proves nonexistence of key k by proving that its predecessor key k' exists and that k is the predecessor of $\text{succ}(k)$.

Given this data structure, a creation proof for a key k involves the proof of its predecessor's membership. Creating a mapping for k given its predecessor k' involves setting $next_k \leftarrow \text{succ}(k')$ and then $next_{k'} \leftarrow k$, or as follows:

```
# key is the key to be inserted
# pred_ctx holds the context for pred(key)
def Put_create(db, key, val, pred_ctx):
    old_pred = pred_ctx.slot
    new_pred = pred_ctx.slot
    new_last = Slot(key=key, val=val,
                    next=pred.slot.next)
    new_pred.slot.next = key
    slot_append(db, new_last)
    slot_write(db, pred_ctx.index, old_pred,
               new_pred, pred_ctx.pf)
```

Now, we can write the following implementation of Put:

```
# db.pred finds a key's predecessor
def Put(db, key, val, ctx):
    if ctx.slot.key == key:
```

```
        Put_modify(db, val, ctx)
    elif ctx.slot.key == db.pred(key):
        Put_create(db, key, val, ctx)
    else:
        raise Exception("invalid ctx")
```

A deletion proof for a key k involves both the proof of k's existence and the proof of its predecessor's value. Removing k given its predecessor k' involves setting $next_{k'} \leftarrow next_k$ and deleting the slot of k. This gives us the following implementation of Delete:

```
# last_ctx holds the context for the last slot
def Delete(db, pf, pred_ctx, last_ctx):
    if pred_ctx.slot.next != pf.slot.key:
        raise Exception("invalid pred_ctx")
    old_pred = pred_ctx.slot
    new_pred = pred_ctx.slot
    new_pred.slot.next = pf.slot.next
    slot_write(db, pred_ctx.index, old_pred,
               new_pred, pred_ctx.pf)
    slot_truncate(db, pf.index, pf.slot,
                  last_ctx.slot, last_ctx.pf)
```

For completeness, we present the implementation of Get:

```
def verify_ctx(db, ctx):
    i, val, pf = ctx.index, ctx.slot, ctx.pf
    bucket, off = i/B, i%B # floor division
    c0 = db.commits[bucket]
    if not Verify(c0, off, val, pf):
        raise Exception("proof is invalid")

# db.zero_value() returns a value denoting an
# absent key
def Get(db, key, ctx):
    verify_ctx(db, ctx)
    if ctx.slot.key == key:
        return ctx.slot.value
    else:
        return db.zero_value()
```

C. Amortizing Cryptographic Overhead

At any point, the last bucket in the dictionary may be not be completely filled. We call this bucket the *tail bucket*, and it exists whenever s is not a multiple of B. Aardvark applies two optimizations on the tail bucket to reduce the cost of creation and deletion operations.

To reduce the cost of creation operations, the vector which backs the tail bucket is not committed to but instead stored directly on all validators. As a result, operations which modify the tail bucket, which include all key insertion operations, do not need to run CommitUpdate on the corresponding commitment.

To reduce the cost of deletion operations, slot deletions from the tail bucket are applied lazily. Validators clear the last slot of the dictionary by decrementing s. Once the last item has been deleted from a tail bucket, and a subsequent deletion request must be processed, the new tail bucket must be initialized, which involves a single full decommitment to the bucket.

Because the bandwidth cost of a single decommitment is relatively high but amortizes well over time, validators cache the preimages of some suffix of the buckets in a *tail cache* and synchronize this cache in the background. If a validator lacks the preimage to a tail bucket, it refuses subsequent deletion requests until it can resynchronize.

Validators forget old buckets as creation operations accumulate, and they request buckets from archives as deletion operations accumulate. To reduce thrashing when creations and deletions happen frequently across a bucket boundary, validators maintain a minimum tail cache size of L but a maximum tail cache size of $2L$.

VI. Concurrent Transaction Processing

The dictionary design described previously allows archives to create proofs authenticating an update from some version of the dictionary to a subsequent version. By definition, the dictionary state changes with the commitment of every (read-write) transaction, invalidating proofs against an older state.

For a slow system, deploying the authenticated dictionary described above is relatively straightforward: for every key updated by a transaction, a corresponding proof is provided that authenticates the update. Since state changes so slowly, it is feasible for a user to recompute invalidated proofs every time a new transaction is confirmed.

However, this design fails to fulfill the needs of high-throughput systems. With the same deployment of our authenticated dictionary, proofs may be invalidated much more quickly, requiring either archives or users to constantly recompute and resubmit proofs. This problem worsens when the system is under heavy load, as the buildup of pending transactions further compounds the cost of recomputing proofs. Moreover, an attacker may degrade service to a key by constantly making small modifications to the bucket-neighbors of a key, invalidating the old proofs held by the user. Unfortunately, ProofUpdate operations are relatively slow in a pairing context (see Table I): updating a single proof requires around two hundred microseconds.

Thus, for the authenticated dictionary to be deployed practically, a *versioning* system must exist. The dictionary should accept transactions with stale proofs even while concurrent operations modify state referenced by that transaction.

Aardvark addresses this problem by eliminating the need to update proofs entirely on the critical path.[4] Instead, Aardvark relies on two main modifications to standard transaction processing to achieve dictionary versioning. First, when constructing a transaction, users declare the minimum and maximum sequence numbers which the transaction may be executed with, and all proofs attached to the transaction refer to the state of the dictionary as of the minimum sequence number. Proofs are valid so long as they are executed at a sequence number between the minimum and maximum numbers (inclusive), and the difference between these sequence numbers must be at most the *maximum transaction lifetime* τ of the system, which is a global constant set when Aardvark is initialized. Second, all validators hold the dictionary digest not as of the most recent transaction number, but instead as of the system state τ transactions ago. For each transaction between the oldest

[4]The only need for ProofUpdate is for archive-free operations, as specified in §VI-E.

state and the newest state, validators maintain a single *delta* on the state (one for each transaction). For each key modified by the transaction, this delta contains both the old value and the updated value of that key.

These two modifications allow validators to efficiently authenticate transactions with varying minimum sequence numbers. Because the difference between minimum and maximum sequence numbers is bounded, any transaction that is seen by a validator is either expired, or it must be no more than τ transactions old. Given a transaction no more than τ transactions old, a validator may use its in-memory deltas to either authenticate the freshness of a proof, or it will invalidate the proof because it observed a more recent change to key state. Since each validator holds τ transactions' worth of updates, it must have seen all such recent changes.

Choosing τ. The choice of τ trades off between the dictionary's asynchronicity and the amount of extra overheads for validators in maintaining the cache. A very small value of τ reduces cache overheads. However, when the dictionary is under heavy load, a client's transaction may expire before it is executed, forcing the client to update the proof before resubmitting the transaction. A larger value of τ allows old transaction proofs to be accepted by the system for a longer period of time, reducing this extra work when the system is congested.

A. Key-Value Pair and Proof Versioning

Aardvark implements its cache of in-memory deltas as follows. First, for every slot modified by a transaction, Aardvark holds *slot deltas*, which contain both the old value and the new value of the slot within a hash table, keyed by slot indexes. Second, for every key that was modified by a transaction, Aardvark maintains *key deltas*, which contain the both the old and the new versions of a value corresponding to a given key. Third, for every bucket that was modified by every transaction, validators in Aardvark maintain *vector commitment deltas* (abbreviated *VC deltas*), which include the old value and new value of every vector commitment to that bucket. Validators also cache a variety of auxiliary state such as the net change in the number of keys and deferred updates to the tail bucket.

Under a versioning system, Aardvark validators can identify when a particular key-value pair given by a proof is stale and update these pair to their latest state. Each of the proofs used for Get, Put, or Delete is either a proof corresponding to a key or a proof corresponding to its predecessor. These proofs are created via a pair of operations supported by archives Read and ReadPred (as described in the previous section):

```
# db is a handle for archive storage
#   db.version is the latest snapshot version
#   db.deltas is a list of key deltas
#   db.slots is the list of key slots
# tau is the maximum transaction lifetime
def LookupSlotAtIndex(db, index):
    version = db.version + tau
    # slot affected recently?
    while version >= db.version:
        delta = db.deltas[version]
        if index in delta.slots:
            # yes; last update in deltas
            return delta.slots[index].new
        version = version - 1
    # no; last update on disk
    return db.slot_at_index(index)

def LookupIndexOfKey(db, key):
    version = db.version + tau
    # key affected recently?
    while version >= db.version:
        delta = db.deltas[version]
        if key in delta.old:
            if key not in delta.new:
                raise Exception("key deleted")
            # yes; index in slot delta
            return search(delta.slots, key)
        version = version - 1
    # no; index on disk
    return db.index_of_key(key)

# search_pred searches a delta for the
# predecessor of a key
def LookupPredOfKey(db, key):
    version = db.version + tau
    # pred(key) affected recently?
    while version >= db.version:
        delta = db.deltas[version]
        pred, i = delta.search_pred(key)
        if pred.next = key:
            # yes; delta contains pred(key)
            return i
        version = version - 1
    # no; pred(key) on disk
    return db.pred_of_key(key)

def ComputeContext(db, index):
    vector = []
    i = index/B # floor division
    top = i + B
    while i < top:
        slot = LookupSlotAtIndex(db, i)
        vector.append(slot)
        i = i + 1
    return Context(index=index,
                   version=db.version,
                   slot=vector[index],
                   pf=Open(vector, index))

def Read(db, key):
    index = LookupIndexOfKey(db, key)
    return ComputeContext(db, index)

def ReadPred(db, key):
    index = LookupPredOfKey(db, key)
    return ComputeContext(db, index)
```

To execute a transaction, validators first execute each Get operation either with a proof of membership or nonmembership:

```
# db is a handle for verifier storage
#   db.version is the latest snapshot version
def LookupVC(db, version, bucket):
    # bucket affected recently?
    while version >= db.version:
        delta = db.deltas[version]
        if bucket in delta.vcs:
            # commitment is in VC deltas
            return delta.vcs[bucket]
```

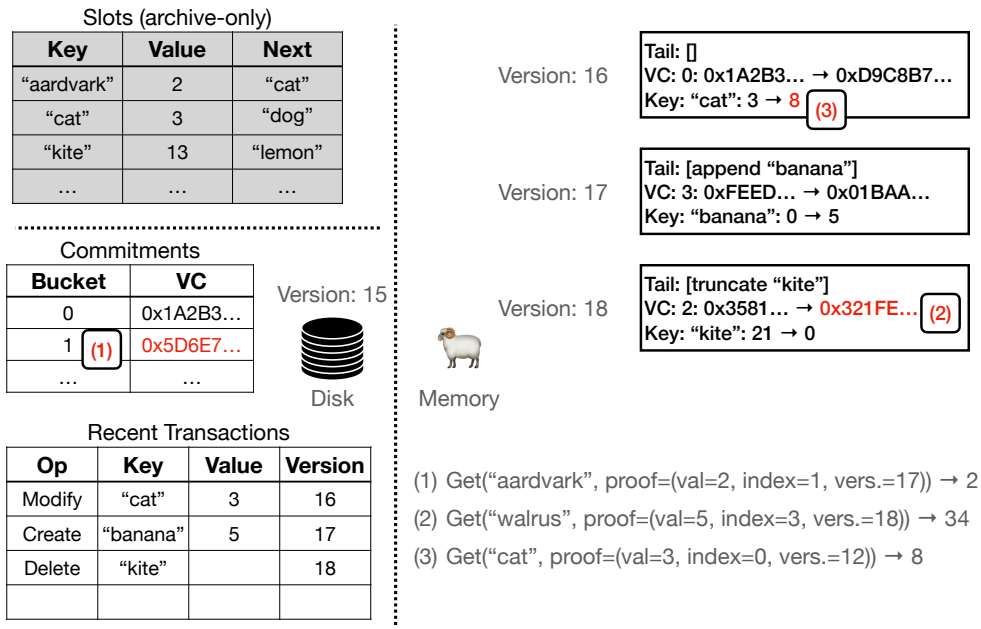

Fig. 3. Versioning in Aardvark with $\tau = 3$. For the first read, Get(“aardvark”), neither the key itself nor its bucket has been modified recently, so the archive's proof is guaranteed to be valid for the vector commitment on the validator's disk. For the second read, Get(“walrus”), the value itself has not been modified recently; however, a deletion operation relocated it. Its proof, which was created after deletion occurred, will be present in a VC delta. For the third read, Get(“cat”), the value was modified recently, so the proof is unnecessary; as a result, the dictionary can simply look up the value present in the key delta.

```
        version = version - 1
    # commitment is on disk
    return db.lookup_vc(bucket)

# ctx is a versioned context for the key
# delta.modified checks whether the delta
# modified the key
# delta.new holds the new value for a key,
# if one exists
def Get(db, key, ctx):
    # key affected recently?
    version = db.version + tau
    while version >= db.version:
        delta = db.deltas[version]
        if delta.modified(key):
            if key in delta.new:
                # key was created/modified
                return delta.new[key]
            # key was deleted
            return db.zero_value()
        version = version - 1
    # no: must find first vector commitment
    # before ctx.version
    bucket = ctx.slot.index/B
    vc = LookupVC(db, ctx.version, bucket)
    if not Verify(vc, ctx.index,
                  ctx.slot, ctx.pf):
        raise Exception("Proof invalid")
    if ctx.slot.key == key:
        return ctx.value
    return db.zero_value()
```

Then, with the updated values, they execute Put and Delete (which additionally expects a key's predecessor). These operations are executed as described previously, except that queries on the current state also traverse the list of deltas.

If a transaction succeeds, validators then compute a new set of deltas on the slots, keys, and vector commitments affected by that transaction, saving the transaction to disk. Once news of a successful transaction arrives at archives and validators, they apply the latest set of vector commitment updates to state on disk (and archives apply slot updates) and then forget the earliest set of deltas. Here, updates are sorted by a consistent ordering so that any validators which execute transactions in the same order arrive at the same sequence of digests.

B. Transaction Expressiveness

Our choice of versioning scheme allows Aardvark transactions to be fairly expressive. Aardvark transactions can implement atomic changes to values such as addition and subtraction, which may be useful for operations such as transfers in a cryptocurrency.

At the same time, our versioning scheme also imposes some limitations within a transaction. For instance, all keys must be determined ahead of time and before a client's initial query to an archive. Aardvark does not optimally support transactions where the choice of one key is dynamic within the transaction. For instance, if an application relies on Aardvark keys to store the nodes of a tree, a transaction which modifies some attribute of a node's parent will be invalidated by an interleaving transaction which modifies the same node's parent.

C. Consistent Proof Validation

In certain applications (such as cryptocurrencies) it is critical for all validators to be in agreement about accepting or rejecting a transaction, as transactions that are seen to be valid by some validators but not others will cause the state of the system to diverge.

The design described above supports consistent lookups with a small modification: searches for a key in some delta must stop τ transaction numbers prior to the current version. So long as τ is consistently known across all validators, this procedure will return consistent results across all validators who see the same latest transaction number.

D. Block Batching

To reduce the in-memory overhead of maintaining deltas and the cost of searching deltas for keys, Aardvark supports batching transactions into *blocks*. Once a block of transactions has been committed, Aardvark merges all deltas in the same block. With a block-commit optimization, versions (and τ) may refer to block sequence numbers, as opposed to transaction sequence numbers, so as to expedite the application of deltas.

Blocks in Aardvark naturally fit with batching optimizations at higher application layers. Moreover, deltas in Aardvark can be made to be hierarchical, so that arbitrary sub-blocks of transactions may be atomically applied to Aardvark state.

E. Archive-free Operations

The dictionary design presented above relies on archives for availability: if the archives go down or refuse to serve a user, the user cannot perform all operations. However, with small modifications, certain operations may be decoupled from archives, which allows the system to continue operating even if all archives go down. We call these operations *archive-free* operations.[5]

Specifically, consider a client which does not store the entire state of the dictionary but does observe all authenticated operations to that dictionary. Suppose that this client possesses a correct proof of that key's value (or a proof of its absence) against some dictionary D. Then every time the dictionary is updated to a new state D' with that update, the client can maintain the freshness of this proof by applying the update to it via SyncContext.

At a high level, SyncContext is implemented as follows. If a client has a current context for a key-value pair, it can update this context when it observes an update to the dictionary. An update to the context involves one of: overwriting (or deleting) the value, overwriting the value of another key belonging to the same bucket, or moving the key into a previously-deleted key's slot. Since all three cases require the presentation of the old contexts of the key's possibly-new bucket, as well as the corresponding deltas, any user following the key's updates will have an updated context after either a Put or a Delete operation by calling ProofUpdate on the vector proof. (If the key itself is deleted, then proof of the key's predecessor given by the Delete operation proves that the key is mapped to the zero value as of the new dictionary state.)

Given this observation, any client that maintains fresh proofs of a key and/or its predecessor can issue arbitrary valid Get, Put, or Delete operations without ever consulting an archive. All of these operations will generally execute on a validator, even if all archives are down. Moreover, this operation only needs to be issued once every τ sequence numbers pass (as opposed to once per transaction).

There is one exception: at a certain point, a validator may run out of its tail cache because too many deletions have happened. In that case, no subsequent deletions may be executed because the tail cannot be moved to fill new slots.

[5]One additional practical benefit of archive-free operations is that they reduce steady-state computational load on archives. These savings can be especially significant scaled over user base, as Open involves a non-trivial computation (§IV).

VII. Evaluation

Our evaluation intends to answer the following questions:

- *Storage Savings*: How much disk space does an Aardvark validator require compared to the archive, which stores all data?
- *Bandwidth Costs*: How large are Aardvark's proofs? What additional networking overhead is introduced by Aardvark?
- *System Throughput*: What are the processing costs required to sustain a given level of system throughput for both validators and archives? How parallelizable are these costs?

We give closed-form expressions for storage and bandwidth costs. For computation costs, we present an empirical evaluation on a prototype implementation of Aardvark. This implementation is available at https://github.com/algorand/aardvark-prototype.

A. Storage and Bandwidth Analysis

Storage Savings. Aardvark requires archives to store all records in the database. If s is the number of key-value pairs, and $|k|$ is the size of a key, and $|v|$ is the size of a value, the archival storage cost for the database is

$$S_A = s(|k| + |v|).$$

We emphasize that in the absence of Aardvark or some other authenticated storage mechanism, this is the storage cost which would be incurred by every validator.

In Aardvark, validators must store a commitment to a bucket every B records. Validators must also store the records in the tail bucket, and all records for all buckets in its cache. This cache is at most $2L$ buckets large (and at least L buckets small, unless the database is less than L buckets large). Moreover, validators must store the last τ transaction blocks to process proofs up to τ blocks stale. If the size of an encoded transaction block is $|T|$ and the size of a vector commitment is $|c|$, this means that the validator storage cost for the database is upper bounded as follows:

$$S_V < \lceil s/B \rceil |c| + (2L+1)(|k| + |v|)B + \tau|T|.$$

A natural choice for $2L$ is a small multiple of $\frac{|T|}{B|t|}$, which allows validators to execute several blocks of continuous deletion requests regardless of the availability of archives. If $\frac{|T|}{|t|} = 10$, then a value of $2L = 30$ is sufficient. Even if blocks are confirmed very quickly (e.g., once a second), setting $\tau = 1000$ allows clients a considerable amount of concurrency (e.g., proofs remain valid for fifteen minutes).

As a result, because the last two terms in the equation above are fairly small (around 1GB), we can drop them from an asymptotic comparison of costs between a validator and an archive. Therefore, as s grows large, the ratio between a validator's cost and an archive's cost in the limit is

$$\frac{S_V}{S_A} \to \frac{B(|k| + |v|)}{|c|}.$$

As described in §IV, $|c| = 48$ bytes in our particular commitment scheme. Choosing a value such as $B = 1000$ for keys of size $|k| = 32$ bytes gives Aardvark validators a savings factor of more than $1000\times$; we note this savings factor increases with the size of $|v|$.

Bandwidth Costs. The bandwidth costs of a transaction consist of the cost of transmitting the transaction plus the costs of transmitting the values for each key in the transaction's key set and the proofs of their correctness. Thus, for each key we transmit $|k| + |v| + |\rho|$ bytes (because we also transmit the successor key). Recall that one proof of key nonmembership is required to insert a new key, one proof of key membership is required to modify ane existing key, and two proofs of key membership are required to delete a key. If an transaction is $|t|$ bytes long, a proof is $|\rho|$ bytes long , and a transaction creates n_1 keys, modifies n_2 keys, and deletes n_3 keys, then the number of bytes transmitted is

$$(n_1 + n_2 + 2n_3)(|k| + |v| + |\rho|) + |t|.$$

Recall that proofs in Aardvark are composed of a version number and an index in the dictionary's slot ordering, in addition to a 48-byte vector commitment proof (see §IV), which means that a size of $|\rho| = 64$ bytes is sufficient for a proof corresponding to a paritcular key. With $|k| = 32$ and $|v| = 8$, this corresponds to roughly 100 bytes of overhead per transaction per Put operation and 200 per Delete operation.

As compared to transaction size, proof overhead is greatest for small transactions that read and write many keys and least for large transactions that read and write few keys. Note that transactions include the key set read and written by the transaction, which also contributes to transaction size.

Note that by aggregating proofs, we can transmit only a single proof, which halves the marginal overhead to roughly 50 additional bytes per transaction per key, plus a fixed cost of transmitting the aggregated proof. Applications where systems involve operations on many different keys benefit substantially from aggregation.

B. System Throughput

We implement our vector commitment scheme in Rust [2], using a pairing library for algebraic operations [1]. (More recent implementation of pairings (such as [21] and [22]) are likely to provide a noticeable speed-up to our benchmarks.) We call into this code from a Go program which implements Aardvark's versioned dictionary.

For crash-safety we use a SQLite database operating under the serializable isolation level . We store in the database vector commitments, recent transactions, and, for archives, key slots. To minimize the effect of I/O latency costs, we use an in-memory database to hold vector commitments on validators and key slots on archives.

TABLE II. TRANSACTION PROCESSING TIME (SECONDS)

Scenario	Machine Role	Cores	Median	Min	Max
10,000 Put (modify)	Archive	1	440	438	441
10,000 Put (create)	Archive	1	437	436	438
10,000 Delete	Archive	1	875	871	878
100,000 Put (modify)	Validator	1	359	334	371
100,000 Put (create)	Validator	1	383	359	427
100,000 Delete	Validator	1	722	688	770
100,000 Put (modify)	Validator	32	53	48	53
100,000 Put (create)	Validator	32	79	75	83
100,000 Delete	Validator	32	118	115	122

To evaluate the computational overhead of Aardvark, which involves transaction validation, we examine the effects of simple transactions which execute either a single Put or a single Delete operation. We separate Put operations into two cases: (1) Put operations that create a new key in the dictionary, which require proofs of nonmembership, and (2) Put operations that modify an existing key in the dictionary, which require proofs of nonmembership. For Delete operations, we pass in bucket preimages as they are needed for vector decommitment. Moreover, we do not aggregate proofs for deletion operations but instead transmit both proofs separately. We benchmark each of these three cases separately and independently.

Each key is a 32-byte random bitstring, and each value is an unsigned integer $v = 8$ bytes in size. For each operation, we pick its key at random from the set of keys (or we generate a random 32-byte string for Put transactions which create keys).

We set the maximum transaction lifetime to $\tau = 10$ blocks. We pick the first valid block number t_0 uniformly at random between block numbers b and $b - \tau$ and its last valid block number between b and $t_0 + \tau$ (where b is the current block number).

We place $|T| = 10000$ transactions in each block. Put transactions are $|t| = 105$ bytes in size, while Delete transactions are $|t| = 87$ bytes in size. Before running each workload, we pre-populate the dictionary with 1 million random key-value pairs and generate $\tau + 2 = 12$ blocks, each with 10 000 transactions of the same type, in order to pre-populate the in-memory deltas and reach steady state.

We conduct our experiment on a `c5.metal` Amazon EC2 machine and use `numactl` to restrict the physical CPUs available to the system.

To exclude networking latency costs, we benchmark a single archive and a single validator separately.

Table II summarizes our results.

Validator Throughput. Validators perform cryptographic proof verification in parallel and verify proofs on all transactions, even those which only affect recently-modified keys, in order to simulate worst-case performance. For a validator, we pre-generate 100 000 transactions, partition them into 10 blocks, and then issue the entire block to the validator. We measure the time taken to validate and apply transactions in each block. We run three trials for each configuration of 1, 2, 4, 8, 16, and 32 cores, and we plot the results in Figure 4.

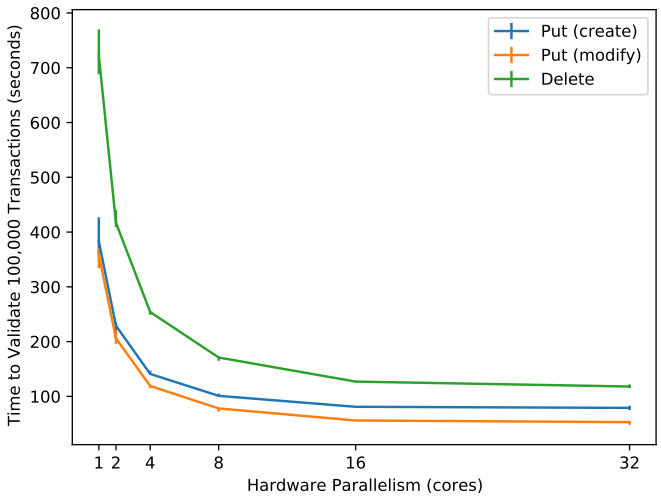

Fig. 4. Scaling of Aardvark relative to number of cores, along with error bars showing the maximum and minimum time taken to validate and apply 100 000 transactions. Adding cores improves performance of proof verification, which is extremely parallel, but updating commitments must still be done in order. We leave batch commitment updates to future work.

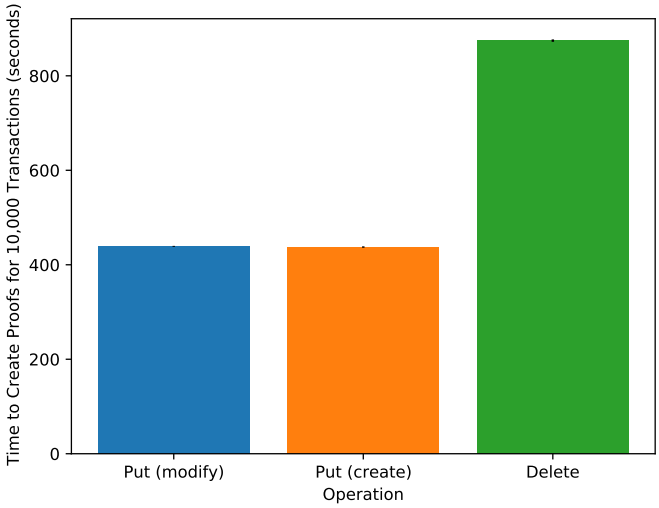

Fig. 5. Time taken by archives to create proofs for 10 000 transactions. Put operations take roughly the same amount of time, regardless of whether they create a new key or modify an existing one. Delete operations require executing both Read and ReadPred, so producing a proof takes twice as long. Note that error bars are present on the figure but are barely visible, as we saw little variability across trials.

Our experiments show that validation requires about thirty seconds of CPU time for blocks filled with Put transactions and over a minute of CPU time for blocks filled with Delete transactions. Since proof verification is expensive, and Delete transactions require two proof verifications while Put transactions require one (in addition to verification of tail slots), processing blocks filled with Delete transactions require twice as much processing time.

Parallelization is beneficial to system throughput, but only up to a point; adding cores yields diminishing returns. Increasing from 1 to 32 cores increases throughput by a factor between 4 and 8. Parallelization produces limited gains for a few reasons. First, because transactions must be serialized, in the worst case, each transaction must be processed in order, one after another. Second, processing all the deltas in each block incurs a fixed cost as the deltas are reconciled with the stable storage. Third, although proof verification is completely parallelized, commitment updates are serialized in our experiments (we did not parallelize commitment updates). Although some degree of parallelism is possible here, an adversary could cause many modifications to affect different indexes of a single bucket, forcing serialization of these updates.

Archive Throughput. Since archival operations are trivially parallelizable, we evaluate archival workloads on a single core. We deliver queries for 10 000 transactions to a single archive and request from it the appropriate proof for each transaction. We run three trials for each workload and plot the results in Figure 5.

Recall that a Put operation which modifies a key requires executing Read, a Put operation which creates a new key requires executing ReadPred, and a Delete operation requires one call to each operation. Our evaluation shows that executing 10 000 Put operations both take roughly the same amount of time (around 7 minutes), while a Delete operation takes roughly twice as much (around 14.5 minutes). This shows that proofs of membership and nonmembership in Aardvark take roughly the same amount of time as they are dominated by the cost of creating the cryptographic proof as opposed to other costs.

VIII. Conclusion

This paper presents Aardvark, an authenticated dictionary suitable for high-throughput, distributed applications. We show that it is possible to create practical authenticated dictionaries with short proofs of membership and nonmembership from pairing-based vector commitments while enforcing tight bounds on extra resource use. We develop a a versioning scheme that enables us to completely ignore expensive proof update costs while supporting concurrent transaction execution. Our evaluation shows that remaining costs reside in proof verification and in updating vector commitments, which benefits greatly from a limited amount of multiprocessing.

Acknowledgment

The authors would like to thank Hoeteck Wee and Adam Suhl for their valuable assistance with the analysis and implementation of vector commitments. The authors would also like to thank Alin Tomescu for useful discussion on the paper's motivation and David Lazar for feedback on a draft of this document.

References

[1] Algorand, "Pairing plus library," 2020, https://github.com/algorand/pairing-plus.

[2] ——, "Source code for pointproofs," 2020, https://github.com/algorand/pointproofs.

[3] M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor, "Checking the correctness of memories," in *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*. IEEE Computer Society, 1991, pp. 90–99, later appears as [4], which is available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.2991. [Online]. Available: http://dx.doi.org/10.1109/SFCS.1991.185352

[4] ——, "Checking the correctness of memories," *Algorithmica*, vol. 12, no. 2/3, pp. 225–244, 1994, available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.2991. [Online]. Available: http://dx.doi.org/10.1007/BF01185212

[5] D. Boneh, B. Bünz, and B. Fisch, "Batching techniques for accumulators with applications to iops and stateless blockchains," pp. 561–586, 2019. [Online]. Available: https://doi.org/10.1007/978-3-030-26948-7_20

[6] V. Buterin, "The stateless client concept," 2017, https://ethresear.ch/t/the-stateless-client-concept/172.

[7] D. Catalano and D. Fiore, "Vector commitments and their applications," in *Public-Key Cryptography - PKC 2013 - 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26 - March 1, 2013. Proceedings*, ser. Lecture Notes in Computer Science, K. Kurosawa and G. Hanaoka, Eds., vol. 7778. Springer, 2013, pp. 55–72, available at http://eprint.iacr.org/2011/495. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36362-7_5

[8] A. Chepurnoy, C. Papamanthou, S. Srinivasan, and Y. Zhang, "Edrax: A cryptocurrency with stateless transaction validation," Cryptology ePrint Archive, Report 2018/968, 2018, https://eprint.iacr.org/2018/968.

[9] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, and E. Gün, "On scaling decentralized blockchains," in *Proc. 3rd Workshop on Bitcoin and Blockchain Research*, 2016.

[10] J. Drake, "Accumulators, scalability of UTXO blockchains, and data availability," 2017, https://ethresear.ch/t/accumulators-scalability-of-utxo-blockchains-and-data-availability/176.

[11] T. Dryja, "Utreexo: A dynamic hash-based accumulator optimized for the bitcoin UTXO set," *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 611, 2019. [Online]. Available: https://eprint.iacr.org/2019/611

[12] M. T. Goodrich, M. Shin, R. Tamassia, and W. H. Winsborough, "Authenticated dictionaries for fresh attribute credentials," in *Trust Management, First International Conference, iTrust 2003, Heraklion, Crete, Greece, May 28-30, 2002, Proceedings*, ser. Lecture Notes in Computer Science, P. Nixon and S. Terzis, Eds., vol. 2692. Springer, 2003, pp. 332–347, available at http://cs.brown.edu/cgc/stms/papers/itrust2003.pdf.

[13] S. Gorbunov, L. Reyzin, H. Wee, and Z. Zhang, "Pointproofs: Aggregating proofs for multiple vector commitments," *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 419, 2020. [Online]. Available: https://eprint.iacr.org/2020/419

[14] B. Libert and M. Yung, "Concise mercurial vector commitments and independent zero-knowledge sets with short proofs," in *Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010. Proceedings*, ser. Lecture Notes in Computer Science, D. Micciancio, Ed., vol. 5978. Springer, 2010, pp. 499–517. [Online]. Available: https://doi.org/10.1007/978-3-642-11799-2_30

[15] R. C. Merkle, "A certified digital signature," in *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, ser. Lecture Notes in Computer Science, G. Brassard, Ed., vol. 435. Springer, 1989, pp. 218–238, available at http://www.merkle.com/papers/Certified1979.pdf. [Online]. Available: http://dx.doi.org/10.1007/0-387-34805-0_21

[16] A. Miller, "Storing UTXOs in a balanced Merkle tree (zero-trust nodes with O(1)-storage)," 2012, https://bitcointalk.org/index.php?topic=101734.msg1117428.

[17] A. Miller, M. Hicks, J. Katz, and E. Shi, "Authenticated data structures, generically," in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, S. Jagannathan and P. Sewell, Eds. ACM, 2014, pp. 411–424, project page and full version at http://amiller.github.io/lambda-auth/paper.html. [Online]. Available: http://doi.acm.org/10.1145/2535838.2535851

[18] C. Papamanthou and R. Tamassia, "Time and space efficient algorithms for two-party authenticated data structures," in *Information and Communications Security, 9th International Conference, ICICS 2007, Zhengzhou, China, December 12-15, 2007, Proceedings*, ser. Lecture Notes in Computer Science, S. Qing, H. Imai, and G. Wang, Eds., vol. 4861. Springer, 2007, pp. 1–15, available at http://www.ece.umd.edu/~cpap/published/cpap-rt-07.pdf. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-77048-0_1

[19] C. Papamanthou, R. Tamassia, and N. Triandopoulos, "Authenticated hash tables based on cryptographic accumulators," *Algorithmica*, vol. 74, no. 2, pp. 664–712, 2016. [Online]. Available: http://dx.doi.org/10.1007/s00453-014-9968-3

[20] L. Reyzin, D. Meshkov, A. Chepurnoy, and S. Ivanov, "Improving authenticated dynamic dictionaries, with applications to cryptocurrencies," in *Financial Cryptography and Data Security - 21st International Conference, FC 2017, Sliema, Malta, April 3-7, 2017, Revised Selected Papers*, ser. Lecture Notes in Computer Science, A. Kiayias, Ed., vol. 10322. Springer, 2017, pp. 376–392. [Online]. Available: https://doi.org/10.1007/978-3-319-70972-7_21

[21] SCIPR-Lab, "Zexe," 2020, https://github.com/scipr-lab/zexe.

[22] Supranational, "blst," 2020, https://github.com/supranational/blst.

[23] P. Todd, "Making UTXO set growth irrelevant with low-latency delayed TXO commitments," 2016, https://petertodd.org/2016/delayed-txo-commitments.

[24] P. Todd, G. Maxwell, and O. Andreev, "Reducing utxo: users send parent transactions with their merkle branches," 2013, https://bitcointalk.org/index.php?topic=314467.

[25] A. Tomescu, I. Abraham, V. Buterin, J. Drake, D. Feist, and D. Khovratovich, "Aggregatable subvector commitments for stateless cryptocurrencies," Cryptology ePrint Archive, Report 2020/527, 2020, https://eprint.iacr.org/2020/527, to appear in SCN 2020.

[26] B. White, "A theory for lightweight cryptocurrency ledgers," 2015, available at https://github.com/bitemyapp/ledgertheory/blob/master/lightcrypto.pdf (see also code at https://github.com/bitemyapp/ledgertheory).